MTL TR 89-94

AD

REPRESENTING KNOWLEDGE INTELLIGENTLY: PRODUCTION RULES, FRAMES, AND TRANSITIONAL NETWORKS

SUZANNE G. W. DUNN
POLYMER RESEARCH BRANCH

October 1989

US ARMY
LABORATORY COMMAND
MATERIALS TECHNOLOGY LABORATORY

U.S. ARMY MATERIALS TECHNOLOGY LABORATORY
Watertown, Massachusetts 02172-0001

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

Mention of any trade names or manufacturers in this report shall not be construed as advertising nor as an official indorsement or approval of such products or companies by the United States Government.

DISPOSITION INSTRUCTIONS

Destroy this report when it is no longer needed.
Do not return it to the originator.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MTL TR 89-94	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* REPRESENTING KNOWLEDGE INTELLIGENTLY: PRODUCTION RULES, FRAMES, AND TRANSITIONAL NETWORKS		5. TYPE OF REPORT & PERIOD COVERED Final Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Suzanne G. W. Dunn		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS U.S. Army Materials Technology Laboratory Watertown, Massachusetts 02172-0001 SLCMT-EMP		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS D/A Project: 1L162105.AH84
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Laboratory Command 2800 Powder Mill Road Adelphi, Maryland 20783-1145		12. REPORT DATE October 1989 13. NUMBER OF PAGES 42
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Artificial intelligence
Natural language
Frame-based knowledge
Production system
Augmented networks
Expert systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(SEE REVERSE SIDE)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

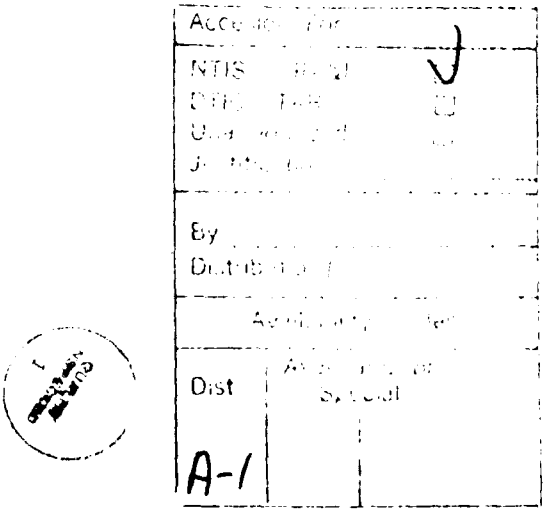
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Block No. 20

ABSTRACT

Designs for a production rule interpreter, a frame-based knowledge system, and an augmented transitional network are described. Knowledge represented in the form of production rules (consisting of domain facts and heuristics) is a good way to model the strong data-driven nature of intelligent action. Using this knowledge, production systems make inferences on the system's current understanding of the "state of the world." The production rule interpreter described uses a bottom-up approach employing a forward-chaining control strategy. Frames are complex data structures for representing stereotyped objects, events, or situations. For intelligent computer programs requiring this type of information, frame based knowledge systems minimize redundant information and may be utilized for acquiring new information which is interpreted in terms of concepts acquired through previous experience. The frame-based system described utilizes procedural as well as declarative information. An augmented transitional network is used for natural language understanding. The transitional network described here uses phrase-structured grammars to syntactically decompose and analyze input sentences.

CONTENTS



INTRODUCTION

Intelligent behavior is not so much due to the methods of reasoning, as it is dependent on the knowledge one has to reason with. In short, intelligence requires knowledge. Knowledge is voluminous, hard to accurately characterize, and constantly changing. Thus, the purpose of knowledge representation is to organize required information efficiently such that the applied artificially intelligent (AI) computer program can readily access that information for decision making, planning, objects/situations recognition, drawing conclusions, and other cognitive functions.[1] This is especially important in the development of expert systems and in natural language understanding.

In AI, a representation of knowledge is a combination of data structures and interpretive procedures that, if used properly in a program, will lead to "knowledgeable" behavior. There are many ways to represent knowledge. Knowledge may be used for acquiring new or additional knowledge, retrieving facts from a knowledge base relevant to the problem, and reasoning about these facts in search of a solution. Representation schemes are traditionally classified into declarative and procedural categories. Declarative categories relate to the representation of facts and assertions, while procedural categories relate to actions. The objective of this report is to describe three programs which were developed to incorporate advanced AI techniques. These techniques are used to represent knowledge declaratively and procedurally for artificially intelligent applications in materials evaluation.

The Production Rule Interpreter Design Section describes a method for representing and interpreting knowledge in the form of "production rules" commonly used in the development of expert systems. The Frame-Based Knowledge Representation System Section describes a frame-based knowledge system which represents the state of the world by associating objects with their respective property lists, including all those properties of the object pertinent to the state description. The Augmented Transitional Network Panser Section illustrates a semantic network approach for describing grammars for natural language interpretation.

All three knowledge representation systems were designed and programmed on a Tektronix 4404 Artificial Intelligence Machine using the Tektronix Franz-Lisp programming language.[2]

A PRODUCTION RULE INTERPRETER DESIGN

System Overview

Production system models accept "knowledge" about a particular problem domain and make "inferences" based on the system's current understanding of the "state of the world." Production systems are a good way to model the strong data-driven nature of intelligent action.[3]

A production system has three primary components: the working memory, the production memory or knowledge, and the rule interpreter. Working memory represents the current state of the system and may be thought of as a temporary account or "short-term memory" of the "facts" known to the system at any particular point in time. The "knowledge" is

1. RICH, E. Artificial Intelligence. McGraw-Hill, Inc., New York, 1983.
2. TEK Programmers Reference. 4400P30 Franz-Lisp, Version 42, Tektronix, July 1985.
3. GEVARTER, W. B. Artificial Intelligence, Expert Systems, Computer Vision and Natural Language Processing. Noyes Publications, New Jersey, 1984.

represented by a set of "production rules" and a set of predefined "predicate functions" from which the rules are constructed. Production rules take the form

<working memory pattern recognized> → <working memory changes>.

For each rule "fired," a situation or state is recognized which leads to the execution of an action causing a change in the situation. The rule interpreter, using a bottom-up/forward-chaining strategy, applies the production rules to the working memory until no further changes in the working memory are observed. The production rule interpreter repeatedly looks for production rules whose left-hand side (LHS) matches working memory. On each cycle, the interpreter picks a rule and does what its right-hand side (RHS) dictates. Since more than one rule may be fired on any cycle, a conflict-resolution strategy is incorporated into the rule interpreter which chooses the "best" rule to fire.

Working Memory

Working memory is a dynamic data structure which always represents the current situation or state of the system at any given time. In this design, working memory is a simple "list" data structure of facts in the form of

((attribute1 value1) (attribute2 value2) ...).

Each fact consists of an attribute name and the current value associated with that attribute. As rules are fired, working memory is altered either by changing current facts (i.e., attribute values are updated) or by adding new facts to memory.

Production Memory/Knowledge

Production memory is a knowledge base of rules which represents the knowledge associated with a particular problem domain. Each rule in the knowledge base is in the form of "situation recognized (i.e., a set of conditions) leads to action executed causing a change in the situation:"

If condition(s) then action(s).

The left-hand side of the rule, "situation recognized," is a Boolean expression of predefined arbitrary conditions or predicates. The right-hand side of the rule, "action" side, is a sequence of predefined actions or functions to be performed. All "conditions" of the rule must be met before the "actions" of the rule will be fired (i.e., executed).

The knowledge base is represented in a "list" data structure where each element of the list represents a unique rule:

((rule1) (rule2) (rule3) ...).

Each rule is represented as an embedded list data structure where the CAR (i.e., the first element of a list) of the rule "list" represents the condition clause of the rule, and the CDR (i.e., a list with the first element removed) of the rule "list" represents the "actions" of the rule:

((condition clause) list of actions).

Each "condition" or "action" in a rule is also represented in a list formatted data structure in the form of [predicate value(s)] where a value can be an attribute, literal value, or a variable.

Condition clauses use logical AND, OR, and NOT functions to "string" conditions on the left-hand side of the rule together:

((AND (condition1) (condition2)) (action1) (action2) ...).

This is interpreted as "If condition1 is true AND condition2 is true, then execute action1, action2, etc."

The logical functions AND, OR, and NOT are predefined in the rule interpreter code; i.e., the LISP built-in logical functions are not used. These logical functions were defined to eliminate the need for the user to quote all attributes and values; i.e., the LISP "AND" function requires that a condition must be in the form of (predicate 'attribute 'value) to execute properly, whereas the rule interpreter defined function "AND" allows the user to write a condition in the form of (predicate attribute value). The Boolean expressions used by the rule interpreter for the construction of the "conditions" of the rules are as follows.

AND (list of conditions)	t if all conditions are true
OR (list of conditions)	t if at least one of the conditions are true
NOT (condition)	t if condition is false (nil)

Predicates and special functions define the language of the rules. All conditions and actions of a rule must be defined in terms of predicates (conditions) and functions (actions). Predicates return true or false, whereas the special functions may have side effects on the data. Both must be defined by the user in a predicate/function definitions file before invoking the rule interpreter.

Predicates

The predefined predicates used for the "conditions" of the rules are as follows.

BIND (variable value)	Binds a variable to a value
EQUAL? (value value)	Tests if two values are equal
(VALUE? (attribute)	Tests if an attribute has been assigned a value

Functions

The predefined functions used for the "actions" of the rules are as follows.

FACT (attribute value)	Assigns a value to an attribute and stores it in working memory
EXECUTE_SCREEN (attribute)	Prompts the user for information about a particular attribute and stores this information in working memory

The functions which are not directly used in defining the rules but are included as a part of the predefined predicates file, "predicates.dat," are as follows.

VARIABLE (value)	Tests if a value is a bound variable
FETCH (variable)	Retrieves the value of a bound variable
ATTRIBUTE (attribute)	Retrieves the value of an attribute

Rule Interpreter

The rule interpreter requires three direct inputs: the initial "state" of the system, the rule knowledge base for the particular problem, and the system "goal." The initial state of the system is the initial situation, or the "facts," the system is given to work with. For example, if a user asks a question of an expert about a particular problem, the user supplies the expert with what he or she knows about the problem. This allows the expert to narrow his or her response. The initial information given to the expert would be analogous to the "facts" given to the system as the "initial state." The "rule knowledge base" is a list of rules which the system uses to solve a particular problem given the initial information or state of the system. The system "goal" is a list of attributes whose values are returned or given as the "answer" to the problem.

The rule interpreter also requires, albeit indirectly, a set of predefined "predicate" functions which are used in the construction of the rules. These predicate functions are the language in which the rules are written. These functions must be defined and loaded in the LISP environment before the system can interpret the rules properly.

The control of rule selection and execution is as follows:

a. All of the rules are examined to see if one or more matches the situation.

b. If only one matches, then the action (RHS) of that rule is executed. If more than one rule matches, the one that is most "specific," defined by a given specificity criteria, is used. If there are two or more rules with the same specificity the one chosen is the earlier one in the user specified rule list.

General Abstract Algorithm

The general abstract algorithm for the rule interpreter is as follows.

```
RULE_INTERPRETER (initial_state, rule_file, predicate_file, goal)

    Initialize variables
    Load rule_file
    Load predicate_file
    Set current_state to initial_state
    Calculate and store the specificity for each rule

    DO UNTIL termination conditions are met
        Find all applicable rules which match the current situation
        Select and execute the most specific rule
```

```
        Store current state of working memory
    END DO
    Print goal value(s) of current_state
END
```

Data Structures

The main data structures which are used to store and maintain specific information used by the rule interpreter are defined as follows.

working_memory	A list of attributes and values which represent the current state of the system
rule_data_base	A list of rules defined by the user
specificity_list	A list of calculated specificities of each rule in the order that the rules appear in rule_data_base
VAR_LIST	A list of bound variables and their values accessed by the functions BIND, FETCH, and VARIABLE
previous_state	Represents the most recent state of the system previous to the current working memory
apply_rules	A list of applicable rules for a given state of working memory

Detailed Abstract Algorithm

A more detailed abstract algorithm is as follows.

```
RULE_INTERPRETER (initial_state, rule_file, predicate_file, goal)

    Initialize all list data structures to nil
    LOAD predicate_file

    SET rule_data_base to (READ_RULES rule_file)
    SET current_state to initial_state
    SET working_memory to initial_state
    SET terminate to nil

    ;Calculate specificity of each rule
    DOLIST rule_data_base
        APPEND (SPECIFICITY rule) to specificity_list
    END DO

    ;Proceed until termination conditions are met
    DO UNTIL terminate is t

        Set previous_state to current_state
        set apply_list to nil
```

```
;Find all applicable rules to current state of the system
DOLIST rule_data_base

      .et conditions to (CAR rule)
      evaluate conditions
      IF conditions are true THEN Append rule to apply_list

END DO

;Order applicable rules according to their specificity
SET apply_list (ORDER_RULES apply_list)

;Execute applicable rules, in order, until state of system changes
DOLIST apply_rules
      evaluate (CAR rule) ;Reevaluate conditions
      DOLIST (CDR rule)
            evaluate action
      END DO
      Reset specificity of rule to a very large number
      IF current_state is not equal working memory THEN return
END DO

SET current_state to working_memory

;Test for termination conditions
SET terminate (TERMINATE_CONDITIONS current_state previous_state)

END DO

PRINT goal from working memory

END
```

In searching for applicable rules for a current "state" of working memory, the condition (LHS) of all rules are evaluated. Once a rule is chosen to fire, the conditions of that rule must be reevaluated before the actions of the rule are executed. This is due to the nature of the built-in BIND function. When the BIND function is used as a predicate in the conditions of a rule, variables are binded to values that may not necessarily be set to the desired value when a rule is chosen to fire. For example, if two applicable rules both BIND the same variable to two different values in their condition clauses, the variable will be set to the value in the most recently evaluated rule, which would not necessarily be the firing rule.

Conflict Resolution

The selection of one rule over all other applicable rules that may "match" the current state of the system depends on the specificity values associated with each rule. The function SPECIFICITY calculates the specificity value of a rule by assigning a numerical value to both the left- and right-hand sides of the rule. The condition (LHS) value is proportional to the length of the expression. The calculation for this length is (number of conditions) + (number of Boolean operators).

The action (RHS) value is proportional to the number of "ask" or system_user communication functions. The numerical value for the RHS of the rule is calculated as 1000*(the number of "ask" functions). The specificities are set much higher for rules which contain the "ask" function(s) over those rules which do not. This is done so that questions will only be asked of the user if the system cannot answer the question via another rule.

The specificity of each rule is calculated as

1000 x (the number of "ask" functions on the RHS) + the LENGTH of the LHS

and stored during the initialization of all variables. This information is sorted so that the specificity of any particular rule is not calculated more than once.

As rules are fired, the specificity of that rule is reset to a very large number. The calculation for resetting the specificity of a rule after it has been fired is (100000 + original specificity). The specificity is reset for two reasons. First, because one of the minimum requirements of the system design is to consider all rules that are applicable for a current state before selecting the most specific one, a "general" rule which may apply in a number of different states or situations may be chosen frequently if it has a low specificity value. Once such a rule is fired, by resetting its specificity to a very large number, the rule will no longer be the most specific of the list. Second, by resetting the specificity value of a rule just fired, and not retiring that rule completely, those rules which must fire more than once to obtain a correct terminal situation will fire, when, and only when, there are no other alternatives in a given situation.

Termination Conditions

Termination conditions are met if, and only if, all applicable rules for a given state have been executed and no changes in the working memory have occurred.

Since there may be several rules which result in the same situation, the termination conditions are not invoked until all applicable rules in a given situation have been executed.

Source Code

The source code is divided into eight major functions: (i) RUN_RULE_INTERPRETER, (ii) RULE_INTERPRETER, (iii) RULE_CONTROL, (iv) ORDER_RULES, (v) SPECIFICITY, (vi) TERMINATION_CONDITIONS, (vii) READ_RULES$, and (viii) EVAL$. This section presents the source code for these eight functions followed by the source code for the logical operators, predefined predicates, and predefined functions discussed previously.

(i) RUN_RULE_INTERPRETER

Function: To run rule interpreter on a given set of initial facts and rules and to then list the values of goal attributes from the final state of working memory.

Input: Initial state of working memory, rule file, predicate definitions file, and a list of goal attributes.

Output: A list of goal attributes and their respective values.

```
(defun RUN_RULE_INTERPRETER (initial_state rule_file predicate_file goal)
    (setq result (RULE_INTERPRETER initial_state rule_file predicate_file)
    (setq goal_ans nil)
    (dolist (g goal)
        (setq goal_ans (append1 goal_ans
            (dolist (fact result)
                (cond ((equal g (car fact))
                    (return fact)))))))
    (setq goal_ans goal_ans))
```

(ii) RULE_INTERPRETER

Function: Transforms an initial state into a final state by interpreting and executing rules.

Input: Initial state of working memory, rule file, and predicate definitions file.

Output: Final state of working memory.

```
(defun RULE_INTERPRETER (initial_state rule_file predicate_file)
    (load predicate_file)

    ;initialize
    (setq VAR_LIST nil) ;list for bound variables
    (setq specificity_list nil) ;list for the specificity of each rule
    (setq rule_data_base (READ_RULES$ rule_file)) ;list of rules
    (setq current_state initial_state)
    (setq previous_state nil)
    (setq terminate nil)

    ;Calculate specificity for each rule
    (dolist (r rule_data_base)
        (setq specificity_list (append1 specificity_list (SPECIFICITY r))))

    ;Do until termination conditions have been met
    (do () ((equal terminate t))
        (setq previous_state current_state)

        ;Select and execute appropriate rules, reset current_state
        (setq current_state (RULE_CONTROL current_state rule_data_base))

        ;Test to see if termination conditions have been met
        (setq terminate (TERMINATE_CONDITIONS current-state previous_state)))

    ;Return final state of working memory
    (setq goal_ans working_memory))
```

(iii) RULE_CONTROL

Function: Selects and executes rules in rule_list where the conditions of the rule selected match the initial_state of working memory.

Input: Initial state of working memory and rules.

Output: Altered state of working memory.

```
(defun RULE_CONTROL (initial_state rule_list)

    ;initialize
    (setq apply_rules nil) ;A list of applicable rules
    (setq apply_rule_num_list nil) ;Rule numbers corresponding to apply-rules
    (setq rcount 0)

    ;Test the conditions of each rule against the state of working memory
    (dolist (current_rule rule_list)
        (setq rcount (+ rcount 1))
        (setq conditions (car current_rule))

        ;Evaluate conditions; if conditions are true store rule
        (cond ((EVAL$ conditions)
            (setq apply_rule_num_list (append1 apply_rule_num_list rcount))
            (setq apply_rules (append1 apply_rules current_rule)))))

    ;Order applicable rules according to their specificities
    (setq apply_rules (ORDER_RULES apply_rules apply_rule_num_list))

    ;Execute the action of each applicable rule (in order) until there is a
    ;change in the state of working memory or there are no more applicable
    ;rules

    (setq rcount -1)
    (setq control_ans
        (dolist (c_list apply_rules)
            (setq rcount (+ rcount 1))
            (EVAL$ (car c_rule)) ;Reevaluate condition of rule
            (dolist (action (cdr c_rule)) ;evaluate each action of rule
                (EVAL$ action))

            ;Reset specificity of rule to a very large number
            (setq sc (- (nth rcount c_list) 1))
            (setq specificity_list (SUBST (+ 100000 sc) sc specificity_list))

            ;Test initial state against current state of working memory
            (cond ((not (equal initial_state working_memory))
                (return working_memory)))))
    ;Return current_state of working memory
    (cond ((equal control_ans nil)
        (setq control_ans initial_state))
    (t (setq control_ans control_ans))))
```

Function SUBST:

Function: To change the specificity of a given rule.

Input: New value, index into list, and list.

Output: Altered list after substitution.

```
(defun SUBST (nvalue indexn nlist)
    (setq alist nil)
    (setq nc -1)
    (dolist (nl nlist)
        (setq nc (+nc 1))
        (cond ((equal nc indexn)
            (setq alist (append1 alist nvalue)))
        (t (setq alist (append1 alist nl)))))
    (setq alist alist))
```

(iv) ORDER_RULES

Function: Orders a list of rules according to their specificity.

Input: A list of rules, and the associated list of rule numbers.

Output: Ordered list of rules.

```
(defun ORDER_RULES (rules count_list)
    (setq spec_list nil)
    (setq oo -1)

    ;Set up list of corresponding specificities
    (dolist (c_rule rules)
        (setq oo (+ oo 1))
        (setq spec_list (append1 spec_list (nth (- (nth oo count_list) 1)
                specificity_list))))

    ;Order rules according to their specificity in ascending order
    (setq ret_list nil)
    (do ((rule rules) (spec spec_list)) ((equal rul nil))
        (setq min_spec 10000)

        ;select minimum specificity
        (dolist (ms spec)
            (setq min_spec (min min_spec ms)))

        ;locate its position in the list
        (setq end_list (member min_spec spec)
        (setq begin_list (ldiff spec end_list))
        (setq nindex (length begin_list))
```

```
        ;store rule and specificity by appending to a new list
        (setq ret_list (append1 ret_list (nth nindex rul)))

        ;delete rule and specificity just stored from original list
        (setq rul (delete (nth nindex rul) rul 1))
        (setq count_list (delete (nth nindex count_list) count_list 1))
        (setq spec (delete (nth nindex spec) spec 1)))

    ;Return ordered list of rules
    (setq ret_list ret_list))
```

(v) SPECIFICITY

Function: Determines the specificity of a rule.

Input: Rule.

Output: An integer corresponding to the length of the LHS of the rule indicating specificity of the rule.

```
(defun SPECIFICITY (rule)

    ;Initialize
    (setq ans 0)

    ;Test RHS of rule for any "ask actions; for each one increase the
    ;specificity by 1000
    (dolist (action (cdr rule))
        (cond ((equal (car action) 'EXECUTE_SCREEN)
            (setq ans (+ ans 1000)))))

    ;Calculate the length of the LHS and add to the specificity
    (setq ans (+ ans (LENGTH (car rule))))

    ;Return calculated specificity
    (setq ans ans))
```

Function LENGTH:

Function: Calculates the length of a list.

Input: List of elements (conditions).

Output: Length of the LHS of the rule.

```
(defun LENGTH (conditions)
    (cond ((equal conditions nil) (setq ret 0))
        ((or (equal (car conditions) 'AND)
            (equal (car conditions) 'OR))
                (setq ret 1)
                (dolist (c (nth 1 conditions))
```

```
                    (setq ret (+ ret (LENGTH c)))))
        ((equal (car conditions) 'NOT) (setq ret 2))
        ((atom (car conditions)) (setq ret 1)))
    (setq ret ret))
```

(vi) TERMINATE_CONDITIONS

Function: Compares the current state of working memory with the last state visited.

Input: Current state of working memory and the last state visited.

Output: t if current state has been visited, nil otherwise.

```
(defun TERMINATE_CONDITIONS (current_state previous_state)
    (cond ((equal current_state previous_state) (return t))))
```

(vii) READ_RULES$

Function: Reads the rules in rule_file into a list, assumes the rules are in list form of conditions → actions.

Input: Name of file where rules are defined.

Output: A list of the rules.

```
(defun READ_RULES$ (rule_file)
    (setq myport1 (fileopen rule_file 'r))
    (setq rule_list (list (read myport1)))
    (zapline)
    (setq test_rule t)
    (do () ((equal test_rule nil))
        (setq test_rule (read myport1))
        (cond ((neq test_rule nil)
            (setq rule_list (append1 rule_list test_rule))
            (zapline))))
    (close myport1)
    (setq ans rule_list))
```

(viii) EVAL$

Function: Evaluates a list of functions and/or predicates that have been predefined.

Input: List of conditions or actions.

Output: t if predicates are true, nil otherwise; or value of function.

```
(defun EVAL$ (others)
    (setq ans (apply (car others) (cdr others))))
```

Logical Operators

The following functions are used for logically stringing rule predicates.

```
(defun AND (others)
    (setq a
        (dolist (next others)
            (setq and_ans (EVAL$ next))
            (cond ((equal and_ans nil)
                (return t)))))
    (cond ((equal a nil) (setq and_ans t))
    (t (setq and_ans nil))))

(defun OR (others)
    (setq o
        (dolist (next others)
            (setq or_ans (EVAL$ next))
            (cond ((equal or_ans t) (return t)))))
    (setq or_ans o)

(defun NOT (others)
    (setq not_ans (EVAL$ others))
    (cond ((equal not_ans t) (setq not_ans nil))
    (t (setq not_ans t ))))
```

Predefined Predicates

The predicates BIND, EQUAL?, and VALUE? are predefined in the file "predicate_file."

(i) BIND

Function: To bind a variable to a value.

Input: Variable and value.

Output: t.

```
(defun BIND (variable value)

    ;If "value" is a variable then set "variable" to its value
    (cond ((VARIABLE value) (setq value (FETCH value))))

    ;If "value" is an attribute then reset the "variable" value to the
    ;attributes value
    (setq val (VALUE? value))

    ;If "value" is not an attribute then take "value" as a literal
    (cond ((equal val nil) (set_variable variable value))
        ;Else bind variable to the value of the attribute
        (t (set_variable variable (nth 1 (ATTRIBUTE value))))))
```

Function set-variable:

Function: Associates a variable with a value and stores it in a list.

Input: Variable and value.

Output: t.

```
(defun set_variable (variable value)
    (setq flag
        (dolist (var_pair VAR_LIST)
            (cond ((equal variable (car var_pair))
                (setq VAR_LIST (subst (list variable value) var_pair
                VAR_LIST))
                (return t)))))
    (cond ((equal flag nil)
        (setq VAR_LIST (append1 VAR_LIST (list variable value)))))

    ;Return t
    (setq ret_ans t))
```

Function VARIABLE:

Function: Tests to see if a value is a variable.

Input: Value.

Output: t if the value is a variable, nil otherwise.

```
(defun VARIABLE (value)
    (setq ans
        (dolist (v VAR_LIST)
            (cond ((equal value (car v )) (return t))))))
```

Function FETCH:

Function: Retrieves the value of a variable from the variable list.

Input: Variable name.

Output: Value of variable, or nil if it doesn't exist.

```
(defun FETCH (variable)
    (setq ret_ans
        (dolist (var_pair VAR_LIST)
            (cond ((equal variable (car var_pair))
                (return (nth 1 var_pair))))))

    ;Return value of variable if exist, else nil
    (setq ret_ans ret_ans))
```

Function ATTRIBUTE:

Function: To test if an attribute exists in working memory.

Input: Attribute name.

Output: The attribute and its value, or nil if not found.

```
(defun ATTRIBUTE (attribute)
    (setq ret_ans
        (solist (w_m working memory)
            (cond ((equal attribute (car w_m))
                (return w_m))))))
```

(ii) EQUAL?

Function: Tests to see if two values are equal; val1 and/or val2 can be an attribute, variable, or literal.

Input: Value1 and value2.

Output: t if the two values are equal, nil otherwise.

```
(defun EQUAL? (val1 val2)

    ;If val1 is a variable set first value to the value of the variable
    (cond ((VARIABLE val1)
        (setq first_val (FETCH val1)))

    ;If val1 is an attribute set first value to the value of the attribute
    (t (setq ans (VALUE? val1))
        (cond ((equal ans nil) (setq first_val val1))
        (t (setq first_val (nth 1 (ATTRIBUTE val1)))))))

    ;If the first value equals val2 then proceed and return true
    (cond ((equal first_val val2)
        (setq second_val val2))

    ;Else test val2 as done previously for val1
    (t (cond ((VARIABLE val2)
        (setq second_val (FETCH val2)))
        (t (setq ans (VALUE? val2))
            (cond ((equal ans nil) (setq second_val val2))
            (t (setq second_val (nth 1 (ATTRIBUTE val2))))))))))

    ;Return t or nil
    (setq ret_ans (equal first_val second_val)))
```

(iii) VALUE?

Function: Tests if an attribute is assigned to a value in working memory.

Input: Attribute name.

Output: t if the attribute has a value, nil otherwise.

```
(defun VALUE? (attribute)

    ;If attribute is a variable, reset attribute to the value of the variable
    (cond ((VARIABLE attribute) (setq attribute (FETCH attribute))))
    (setq ans (ATTRIBUTE attribute))
    (cond ((equal ans nil) (setq ret_ans nil))
          ((equal (nth 1 ans ) nil) (setq ret_ans nil))
          (t (setq ret_ans t))))
```

Predefined Functions

The functions FACT and EXECUTE_SCREEN are also predefined in the file "predicate_file."

(i) FACT

Function: Stores a "fact" in the form of (attribute value) in working memory.

Input: Attribute and value.

Output: t.

```
(defun FACT (attribute value)

    ;If "value" is a variable then retrieve the value of that variable
    (cond ((VARIABLE value) (setq val (FETCH value)))
    (t (setq val value)))

    ;Test if attribute is defined in working memory
    (setq w_m (ATTRIBUTE attribute))

    ;Replace old value of attribute in working memory with new value
    (cond ((neq w_m nil)
          (setq working_memory (subst (list attribute val) w_m working_memory)))
    (t (setq working_memory (append1 working_memory (list attribute val)))))

    (setq ret_ans t))
```

(ii) EXECUTE_SCREEN

Function: To prompt the user with a query in menu format.

Input: The attribute which indicates the appropriate query.

Output: t.

Side Effect: Inserts users answer(s) into working memory.

```
(defun EXECUTE_SCREEN (node)

    ;If node is a variable then reset node, and the value of that variable
    ((cond ((VARIABLE node) (setq node (FETCH node))))

    ;Execute screen "node" and store users answer(s) to given question(s)
    (setq user_answers (user_inter node))
    (dolist (fact_list user_answers)
        (cond ((neq (nth 1 fact_list) 'unknown)
            (FACT (car fact_list) (nth 1 fact_list)))))

    (setq ret_ans t))
```

The function EXECUTE_SCREEN is a menu-formatted query in the form of a question and a list of possible answers to which the user selects one via a mouse. This function calls on a program, USER_INTER, which actually prompts the user with the appropriate "screen" and returns the user's answer. A program, PROG_INTER, allows the user to define menu screen queries and associate a key word or "attribute" with each screen. The source code for these two modules is not included in this report.

A FRAME-BASED KNOWLEDGE REPRESENTATION SYSTEM

System Overview

A frame-based knowledge network is a method for representing knowledge in a hierarchical structure within which new data are interpreted in terms of concepts acquired through previous experience.[4] A frame is used to describe a class of objects or an instance of an object. Each frame consists of a collection of slots that describe aspects of that object or instance. Frames are linked in taxonomies using "member links" for class membership and "subclass links" for class containment or specialization. For any given frame, information may be inherited from any one of its ancestor frames, eliminating the need to duplicate information which is inherited. For example, a class of objects can inherit properties from its superclass.

A slot within a frame consists of an attribute and a list of four distinct value parameters which are used to describe that attribute. These attributes also describe attributes in subsequent children frames. The first three value parameters are used to represent the attribute's value, default value, or the procedure to obtain the value of that attribute. The fourth value parameter is a procedure for performing some function if, and only if, a value for the slot's attribute has been modified. Any slot may have one or more of these four value parameters associated with its attribute, specified either within that frame's slot or within the slot of an ancestor's frame.

A frame's attribute may have up to three methods of determining its value. This necessitates choosing from among these values which takes precedence in a given frame-based network. Strategy parameters are used to direct the search for information within a frame-based network. A strategy will direct how and what information or knowledge is retrieved from a network.

4. The Handbook of Artificial Intelligence. William Kaufmann, Inc., Avron Barr, and Edward A. Feigenbaum, ed., v. 1, 1981.

In developing a method for building and accessing frame-based representation of knowledge, specialized routines were developed. These routines include: (i) FGET for accessing and retrieving information from a frame network using various search strategies, (ii) FCREATE and FPUT for building and adding new knowledge to a frame network, and (iii) FSETPAR and FGETPAR for setting and retrieving search strategies in which inheriting information from "parent" frames become necessary.

The next two topic areas will further clarify the structure of the frame-based knowledge system. The first topic area will deal with each individual subcomponent of the system and the second topic area will deal with the appropriate source code.

Frame Network

The frame network is stored in a data file and loaded into memory when required. This network is represented by a "list" of frames with the following format:

(frame1 frame2 frame3 ...).

Frames

Frames are composed of several independent information elements. The first of these is the name of the frame, or object, being defined. Frames also carry information to classify the frame as either generic (concept) or individual (instance). Information is also included about a frame's immediate "parent," specifically whether it is classified as a subconcept of, or an instance of, its ancestor. Lastly, a frame carries a list of its associated "slots" which are its attributes and corresponding value parameters. Each frame is also represented as a "list" with the following format:

(name type parent (slots)).

Frame Attributes

Each frame consists of any number of slots which describe the frame's attributes. Each slot consists of a list of the "attribute" name and four associated values: (i) an absolute value, (ii) a "default" value, (iii) an "If-needed" procedure, and (iv) an "If-added" procedure. If the value, default, If-needed, or If-added values/procedures are not known or do not exist then "nil" is placed in their respective locations within the slot,

slot = (attribute value default if-needed if-added).

The CDR of a slot will be labeled as the "list of values" that an attribute may have.

Strategy Parameters

The strategy contains two parameters, the search strategy for retrieving information and a set of four flags which indicate which of the value parameters may be used. These parameters are represented with a "list" data structure in the following format:

((strategy) (flags)).

The set of flags are represented as a list of four numbers (0/1), signifying whether or not that value parameter of an attribute may be used. The location of each value in the list of flags corresponds to the position of that value in the list of value parameters within a slot; i.e., the CDR of a slot. The flags are represented as 0 or 1, where a 0 represents "off" and a 1 represents "on." "Off" value parameters cannot be accessed regardless of the search strategy. For example,

(1 1 1 1) → all value parameters are accessible,

(1 0 1 1) → no default values may be used.

Recalling that the first three value parameters are used in determining the value of an attribute, the search strategy is represented as a list of numbers (0, 1, or 2) and nil elements. This search strategy is used to determine the sequence in which the first three value parameters (absolute, default, or if-needed) will be retrieved as an attribute's "value." The numbers 0, 1, and 2 correspond to the location of the first three value parameters in the CDR of a slot. The nil elements are used to indicate when to access the parent frame. This is clarified in the following example.

Frame1					**Frame2**				
generic					individual				
parent	nil				parent	Frame1			
attribute1	5	nil	nil	nil	attribute1	nil	4	nil	nil
attribute2	nil	6	nil	nil					

Using a list data structure, these two frames would be represented as

(Frame1 generic nil ((attribute1 5 nil nil nil) (attribute2 nil 6 nil nil)))

(Frame2 individual Frame1 ((attribute1 nil 4 nil nil))),

respectively. If the strategy parameters are ((0 1 2 nil) (1 1 1 1)), then the value for Frame2's attribute1 would be 4. This is accomplished by first looking for an absolute value, then a default value, then an if-needed value, before trying an ancestry frame. However, if the strategy parameters were ((0 nil 1 nil 2 nil) (1 1 1 1)), then the value for Frame2's attribute1 would be 5; i.e., look for an absolute value all the way up the hierarchy of ancestry frames (represented by the nil after the 0) before looking for any default or if-needed values.

By representing the strategies in this way, the user may design any strategy and is not limited to predefined ones.

Algorithms/Source Code

A frame-based network is a methodology for representing information. Therefore, the programs represented here are somewhat separate routines to be used independently or within some other system (for example in a production rule interpreter).

Data Structures

The following data structures are commonly used for the FGET, FPUT, FCREATE, FSETPAR, and FGETPAR routines.

frame_network	the file name of where the frame network is stored
frame	the name of the "frame" of interest
attribute	the name of the "attribute" of interest
parameters	the strategy parameters and flag settings
strategy	the strategy (CAR parameters)
flags	the flag settings (nth 1 parameters)
value	the value to be stored for an attribute
cur_frame	the current frame list
slot	the current attribute slot list
values	the list of attribute values (CDR slot)
parent	the immediate ancestor or cur_frame

Retrieving Information (FGET)

The FGET routine makes full use of the strategy parameters to make choices between the absolute value, default value, or if-needed procedure when inquiring about a particular frame and attribute. Information about the 'frame' and 'attribute' of interest are stored as global parameters which may need to be used in an 'if-needed' procedure.

Abstract Algorithm: (FGET frame_network frame attribute)

```
load 'parameters' and 'network' from file 'frame_network'
assign global variables
strategy = (car parameters)
flags = (nth 1 parameters)

**break up strategy into substrategies so that each substrategy ends with "nil" (next
frame up) and assign 'strategy_list' to the list of these substrategies. Note: if the
appropriate flag is not set (1) do not include the strategies.

For each substrategy in 'Strategy_list' Do until a value other than nil is found or the
network has been exhausted.

    * cur_frame = find 'frame
      If cur_frame = nil then return nil
      Else
        slot = find 'attribute in cur_frame
        parent = (nth 2 cur_frame)
        If slot = nil then
            If parent = nil then return nil
            Else
```

```
                frame = parent
                goto *
        Else
            values = (cdr slot)

            Dolist substrategy (binding s to its elements)
                If (nth s values) ≠ nil then
                    return (nth s values) or
                    return (eval (nth s values))
            If no value found then
                frame = parent
                goto *

    END
```

The FGET routine is divided into two functions, FGET and FGET_FRAME. FGET does all necessary initialization and oversees strategy execution. In this first function, the search strategy is broken down into substrategies due to the affected use of the nil construct. This allows inheritance to occur for each substrategy independently and in the proper sequence. FGET_FRAME is a recursive procedure which actually handles all inheritance routines for each substrategy and called on by FGET. The following example clarifies this.

Search Strategy = (0 nil 1 2 nil)

Flags = (1 1 1 1)

FGET would divide the search strategy into *two substrategies*;

((0) (1 2))

FGET calls on FGET_FRAME to search through the Frame's ancestry hierarchy for an absolute value first. If there exists no such value, FGET reexecutes FGET_FRAME to search the hierarchy for any default or if-need values, respectively. If

Flags = (1 1 0 1)

then the substrategies would be ((0) (1)) since the flag for the if-needed value is not set.

Function FGET:

Function:	To retrieve the value of an attribute of a given frame.
Input:	Network file, frame, attribute.
Output:	Value associated with "frame" and "attribute" in frame_network.
Side Effect:	Executes if-needed procedures associated with "attribute" at any level of inheritance.

```
(defun FGET (frame_network frame attribute)
```

```
;load network into memory
(setq myport (infile frame_network))
(setq parameters (read myport))
(setq network (read myport))
(close myport)

;set global variables
(setq global_frame frame)
(setq global_attribute attribute)

;set up strategy
(setq strategy (car parameters))
(setq flags (nth 1 parameter))
(setq strategies nil)
(setq s nil)

;break up strategy into a list of substrategies
(dolist (str strategy)
     (cond ((equal str nil) ;nil indicates next frame up
          (cond ((neq s nil)
               (setq strategies (append1 strategies s))))
          (setq s nil))
     (t (cond ((equal (nth str flags) 1)
          (setq s (append1 s str)))))))

;return appropriate answer
(setq answer
     (dolist (s strategies)
          (setq ans (FGET_FRAME network frame attribute s))
          (cond ((neq ans nil) (return ans))))))
```

FGET Frame:

Function: Recursive program to retrieve inherited value for "attribute."

Input: Network list, frame, attribute, strategy.

Output: Value of attribute or nil.

```
(defun FGET_FRAME (network frame attribute strategy)

     ;find frame in network
     (setq cur_frame
          (dolist (fn network)
               (cond ((equal (car fn) frame) (return fn)))))

     (cond ((equal cur_frame nil) (setq ret_ans nil)) ;if no frame return nil
     (t
          ;find the slot with the name of attribute
          (setq slot
```

```
            (dolist (s (nth 3 cur_frame))
                (cond ((equal (car s) attribute) (return s)))))

    ;save the name of the frame's parent
    (setq parent (nth 2 cur_frame))

    ;if the slot attribute does not exist then...
    (cond ((equal slot nil)
        (cond ((equal parent nil) (setq ret_ans nil)) ;if no parent return nil

        ;Else try to inherit the value from the frame's parent
        (t (setq ret_ans
            (FGET_FRAME network parent attribute strategy)))))

    ;if the slot exists then...
    (t ;save value default if-needed and if-added for attribute
        (setq values (cdr slot))

        ;try to find value at current frame according to strategy
        (setq ret_ans
            (dolist (s strategy)
                (setq test_value (nth s values))
                (cond ((neq test_value nil)
                    (cond ((> s 1) (return (eval test_value)))
                    (t (return test_value)))))))

        ;if no value is found inherit value if possible
        (cond ((and (equal ret_ans nil) (neq parent nil))
            (setq ret_ans
                (FGET_FRAME network parent attribute strategy))))))))
    (setq ret_ans ret_ans))
```

Storing Information (FPUT)

The FPUT routine does not make use of the search strategy and only uses the 'if-added' flag for determining if the 'if-added' procedure should be executed.

Abstract Algorithm: (FPUT frame_network frame attribute value)

load 'parameters' and 'network' from file 'frame_network'
assign global variables

cur_frame = locate 'frame
slot = locate 'attribute in 'frame

If slot = nil then add new slot to cur_frame
Else replace "nil" or old value with new value

Replace old frame with revised frame in network

If if-added flag is set then

```
        ;look for if_added procedures
        * frame = parent
        If frame = nil then return
        Else
            cur_frame = locate frame
            slot = locate 'attribute
            If slot ≠ nil then
                If if-added procedure exists in slot then
                    execute procedure
                    return
                Else goto *
            Else goto *
    END
```

The FPUT routine is also divided into two functions. The first function handles all initialization and the storage of the user-specified information to include the absolute values of attributes in given frames. The second function handles the search and execution of any if-added procedures which must be executed as a result of the insertion of new information.

Function FPUT:

Function: Inserts a value for a given attribute of a given frame.

Input: File name where network is stored, frame, attribute, value.

Output: Nil.

Side Effect: Executes if_added procedures when appropriate.

```
(defun FPUT (frame_network frame attribute value)

    ;load network into memory
    (setq myport (infile frame_network))
    (setq parameters (read myport))
    (setq network (read myport))
    (close myport)

    ;store global variables
    (setq global_frame frame)
    (setq global_attribute attribute)
    (setq global_value value)

    ;find specified frame
    (setq cur_frame
        (dolist (fn network)
            (cond ((equal (car fn) frame) (return fn)))))

    ;locate specified slot named attribute
    (setq slot
```

```
        (dolist (s (nth 3 cur_frame))
            (cond ((equal (car s) attribute) (return s)))))

    ;if attribute does not exist append the attribute and value to the frame
    (cond ((equal slot nil)
        (setq s
            (append1 (nth 3 cur_frame) (list attribute value nil nil nil)))
        (setq f (subst s (nth 3 cur_frame) cur_frame))
        (setq network (subst f cur_frame network)))

    ;else set/replace old value with new value
    (t    (setq f (subst (list attribute value (nth 2 slot) (nth 3 slot)
            (nth 4 slot)) slot cur_frame))
        (setq network (subst f cur_frame network))))

    ;store in file
    (setq myport (infile frame_network))
    (setq p (read_myport))
    (close myport)
    (setq myport (outfile frame_network 'w))
    (format myport "~a~%~a~%" p network)
    (close myport)

    ;look for if_added procedures that should be executed due to the insertion
    (FPUT_IFADDED network parameters frame attribute nil))
```

FPUT IFADDED

Function: Executes if_added procedures, as appropriate.

Input: Network, parameters, frame, attribute, if-added flag or nil.

Output: Nil.

Side Effect: Executes procedures as appropriate.

```
(defun FPUT_IFADDED (network parameters frame attribute flag)

    ;locate frame
    (setq cur_frame
        (dolist (fn network)
            (cond ((equal (car fn) frame) (return fn)))))

    ;locate slot
    (setq slot
        (dolist (s (nth 3 cur_frame))
            (cond ((equal (car s) attribute) (return s)))))

    ;test if_added flag
    (cond ((equal flag nil)
```

```
        (setq flags (nth 1 parameters))
        (setq flag (nth 3 flags))))

    ;if set then...
    (cond ((equal flag 1)

        ;if no slot exist test ancestors
        (cond ((equal slot nil)
            (setq parent (nth 2 cur_frame))
            (cond ((neq parent nil)
                (FPUT_IFADDED network parameters parent attribute flag))))

        ;Else test if_added position
        (t (setq if_added (nth 4 slot))

            ;if if_added position is nil then look at ancestors
            (cond ((equal if_added nil)
                (setq parent (nth 2 cur_frame))
                (cond ((neq parent nil)
                    (FPUT_IFADDED
                        network parameters parent attribute flag))))

            ;else evaluate procedure
            (t (eval if_added)))))))
```

Creating New Frames (FCREATE)

The FCREATE routine uses the FPUT routine when inserting the value for each attribute of the new frame. This ensures that if the new frame is a descendant of a frame which has an attached 'if-added' procedure to a given attribute, then it will be executed as appropriate (i.e., if, and only if, the if-added flag is set).

Abstract Algorithm: (FCREATE frame_network frame type parent slots)

```
    load 'parameters' and 'network' from file 'frame_network'
    assign global variables
    add new frame to network replacing the value of each attribute with nil
    store new network in file

    For each slot Do
        (FPUT frame_network frame attribute value)
    END DO
END
```

Function FCREATE:

Function: To create a new frame in a given network.

Input: Network file, frame type parent slots.

Output: Nil.

```
(defun FCREATE (frame_network frame type parent slots)

    ;load in network
    (setq myport (infile frame-network))
    (setq p (read myport))
    (setq n (read myport)
    (close myport)

    ;insert and store frame
    (setq c-frame (list frame type parent))
    (setq ss nil)

    ;load everything except the values of the slot attributes
    (dolist (s slots)
        (setq ss (append1 ss (list (car s) nil (nth2 s) (nth3 s) (nth 4 s)))))

    ;save new network
    (setq c_frame (append1 c_frame ss))
    (setq n (append1 n c-frame))
    (setq myport (outfile frame_network 'w))
    (format "~a~%~a~%" p n)
    (close myport)

    ;fput all values for each attribute so that any appropriate if_added
    ;procedures may be executed
    (dolist (s slots)
        (setq value (nth 1 s))
        (cond ((neq value nil)
            (setq attribute (car s))
            (FPUT frame_network frame attribute value)))))
```

Strategy Parameters (FSETPAR and FGETPAR)

The parameters of a network are stored in the network file along with the network. These parameters are used as the default strategy and flag settings so that the user does not need to set them with every FGET, FPUT, and FCREATE routine call. To retrieve these values, the FGETPAR routine may be used. To reset these values, the FSETPAR routine may be used.

Function FSETPAR:

Function:	Sets strategy and flags for the execution of defaults, if-added, and if-needed values.
Input:	Network file, strategy, flags.
Output:	Nil.
Side Effect:	Stores parameters in file 'frame_network.

```
(defun FSETPAR (frame_network strategy flags)
```

```
;load in file
(setq myport (infile frame_network))
(setq test (read myport))
(setq test (read myport))
(close myport)

;store new parameter values
(setq myport (outfile frame_network 'w))
(setq param (list strategy flags))
(format myport "~a~%" param)
(format myport "~a~%" test)
(close myport))
```

Function FGETPAR:

Function: Retrieves current parameter values (strategy and flags).

Input: Network file name.

Output: A list of strategy and flags.

```
(defun fgetpar (frame_network)
    (setq myport (infile frame_network))
    (setq parameters (read myport))
    (close myport)
    (setq parameters parameters))
```

AN AUGMENTED TRANSITIONAL NETWORK PARSER

System Overview

An augmented transitional network (ATN) employs methods used in recognizing natural language efficiently. The system is designed to accept a "grammar" definition and a dictionary of acceptable words which will be used to interpret sentence structures. The ATN interpreter is designed independent of the grammar and dictionary specifications to allow diversity in intelligent applications which require natural language understanding.

The ATN interpreter requires three direct inputs: the grammar definition, the dictionary, and the sentence to be parsed for interpretation. The grammar definition takes the form of a directed network where the nodes of the network are the 'states' and the arcs of the network are traversed if the arc corresponds to the current part of the input sentence being parsed. The dictionary contains a list of words and associated categories.

Registers are used to store parsed words or phrases. These registers are in turn used to interpret the input sentence. The registers are user defined as part of the grammar definition. Built-in functions exist in the ATN interpreter to manipulate user-defined registers as the grammar requires. The user may utilize final register values for obtaining the desired interpretation of an input sentence. This allows the ATN interpreter to be independent of the application.

ATN Network Grammar Definition

The ATN network uses phrase-structured grammars to syntactically decompose an input sentence. The grammar definition is represented with a directed network consisting of a main network and any number of subnetworks, as illustrated in Figure 1. Each network contains a set of nodes and a set of directed arcs connecting these nodes. The nodes of a network represent the "states" and the arcs represent the parts of the language which form the grammar. The starting node of the main network must be labeled with an 'S.' All remaining nodes of this network and any subnetwork are labeled arbitrarily by the user. Arcs of a network are traversed as parts of the language recognized in the input sentences. Arcs are labeled with predefined predicates. These predicates are functions used to perform tests on the input sentences. All networks or subnetworks are terminated with a POP arc. Traversing a POP arc returns control to its parent network.

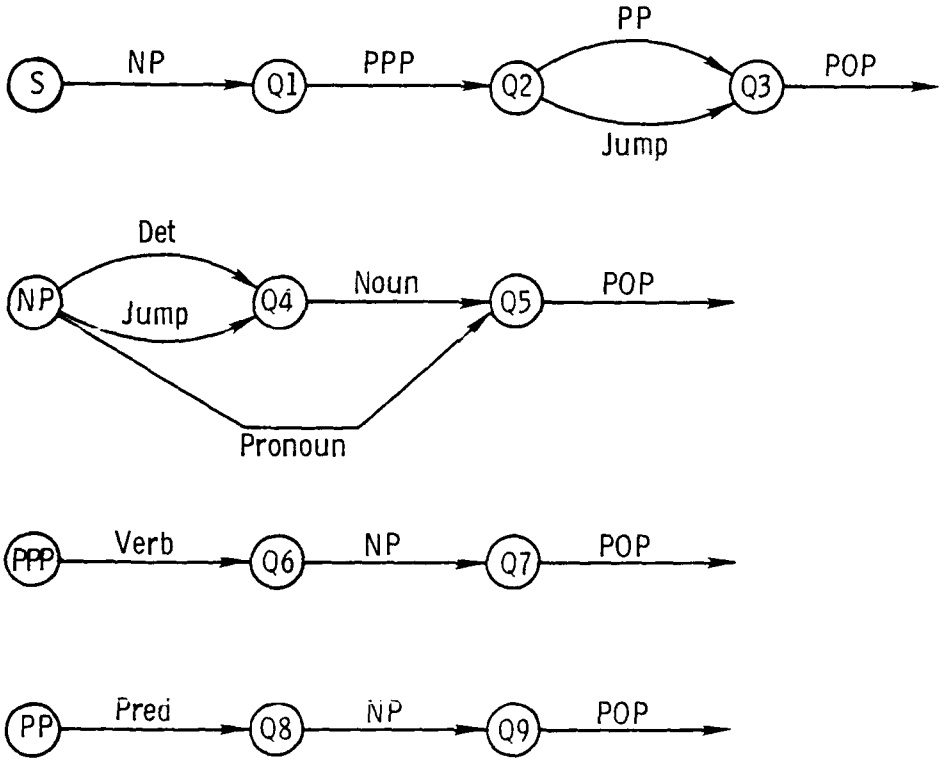


Figure 1. Grammar definition.

Data Structures

The data file which contains the grammar definition consists of a list data structure of nodes and their associated arcs in the following format:

(S arc1 arc2 ... arci)

(nodei arc1 arc2 ... arcj).

Each arc associated with a particular node is represented by a sublist which contains the following information:

(test pre_action command post_action next_node).

Where:

test: A function which must be true in order to traverse the arc.

pre_action: A function call executed before the arc is traversed, usually a register manipulation.

command: A predefined predicate (arc label) which determines if the traversal of the arc is successful or not.

post_action: A function call executed if, and only if, the arc was traversed successfully.

next_node: The label of the node to be traversed next.

The predefined predicates which are used as arc labels are as follows:

CAT (category): Tests the current input work against the dictionary to see if it is a member of a certain category (i.e., noun, verb, predicate, etc.).

WRD (word): Tests the current input word against the given 'word' to see if they are equal.

MEM (wordlist): Tests the current input word against the given 'wordlist' to see if it is a member of that list.

JUMP (): Jumps to the next node.

NUMBER (): Tests if the current input word is a number in numerical representation.

PUSH (startnode): Jumps to the subnetwork with the first node labeled 'startnode.'

POP (registerlist): Returns the previous network with the values of the registers in 'registerlist.'

For example, the grammar in Figure 1 would be represented in a list data structure as follows.

```
        (dolist (w1 word_list)
            (cond ((equal word w1) (return word)))))))
    (cond ((neq ans nil) (setq word_index (+ word_index 1))))
    (setq ans ans))
```

(ii) WORD

Function: Tests if the current word is equal to 'input_word.

Input: The word.

Output: The word if t, nil otherwise.

```
(defun WRD (input_word)
    (setq word (nth word_index user_input)) ;get next word of input sentence
    (cond ((equal input_word word)
        (setq word_index (+ word_index 1))
        (setq ans word))
    (t (setq ans nil))))
```

(iii) MEMBER

Function: Test the current word against a list of words.

Input: A list of words.

Output: The word if a member of the word_list, nil otherwise.

```
(defun MEM (word_list)
    (setq word (nth word_index user_input)) ;get next word of input sentence
    (setq ans
        (dolist (w1 word_list)
            (cond ((equal w1 word)
                (setq word_index (+ word_index 1))
                (return word))))))
```

(iv) JUMP

Function: To jump from one node to another.

Input: Nil.

Output: Nil if no words left in put sentence, t otherwise.

```
(defun JUMP ()
    (setq word (nth word_index user_input) ;get next word of input sentence
    (cond ((equal word nil (setq ans nil))
    (t (setq ans t))))
```

```
(S (t nil (PUSH 'NP) (set_register 'subject *) Q1))

(Q1 (t nil (PUSH 'PPP) (set_register 'prep *) Q2))

(Q2 (t nil (PUSH 'PP) (set_register 'pred *) Q3)
    (t nil (JUMP) nil Q3))

(Q3 (t nil (POP (list 'subject 'prep 'pred)) nil nil))

(NP (t nil (CAT 'det) nil Q4)
    (t nil (CAT 'pnoun) (set_register 'noun *) Q5)
    (t nil (JUMP) nil Q4))

(Q4 (t nil (CAT 'noun) (set_register 'noun *) Q5))

(Q5 (t nil (POP (list 'noun)) nil nil))

(PPP (t nil (CAT 'verb) (set_register 'verb *) Q6))

(Q6 (t nil (PUSH 'NP) (set_register 'NP *) Q7))

(Q7 (t nil (POP (list 'verb 'NP)) nil nil))

(PP (t nil (CAT 'pred) (set_register 'p *) Q8))

(Q8 (t nil (PUSH 'NP) (set_register "NP *) Q9))

(Q9 (t nil (POP (list 'p 'NP)) nil nil))
```

Note that, in this example, all 'tests' are true, there are no 'pre-actions,' and most of the 'post_actions' are register manipulations which are discussed following the ARC predicate functions.

ARC Predicates

The program code for the predefined arc predicates CATEGORY, WORD, MEMBER, JUMP, NUMBER, PUSH, and POP are as follows.

(i) CATEGORY

Function: Tests if the current word is of the appropriate category.

Input: Name of category.

Output: The word if t, nil otherwise.

```
(defun CAT (category)
    (setq word (nth word_index user_input)) ;Get next word of input sentence
    (setq word_list (gethash category dictionary))
    (cond ((equal word_list nil) (setq ans nil))
    (t (setq ans
```

(v) NUMBER

Function: Tests if the current input word is a number.

Input: Nil.

Output: The number if t, nil otherwise.

```
(defun NUMBER ()
    (setq word (nth word_index user_input))
    (cond ((numberp word)
         (setq word_index (+ word_index 1))
         (setq ans word))
    (t (setq ans nil))))
```

(vi) PUSH

Function: Transfer control to a given network.

Input: Starting node of network.

Output: Post action.

```
(defun PUSH (start_node)

    ;store place and registers in current network
    (push_g_stack stack post_act next_node)
    (setq stack nil)
    (setq node (get_node start_node)) ;get node and associated arcs
    (setq arcs (cdr node))
    (setq return_ans

        ;Do until all arcs of start_node have been exhausted or a suitable
        ;answer is found
        (do () ((equal arcs nil))

            ;store all but the first arc in a stack for backtracking
            ;purposes
            (cond ((> (length arcs) 1)
                 (push_stack registers work_index (cdr arcs))))

            (setq cur_arc (car arcs)) ;current arc to be traversed
            (setq test (car cur_arc)) ;parse the arc list
            (setq pre_act (nth 1 cur_arc))
            (setq command (nth 2 cur_arc))
            (setq post_act (nth 3 cur_arc))
            (setq next_node (nth 4 cur_arc))

            ;evaluate test and proceed with arc if t
            (cond ((eval test)
```

```
                    (eval pre_act)
                    (setq answer (eval command))

                    ;if arc is successfully traversed evaluate post action
                    (cond ((neq answer nil)
                         (setq post_act (subst (quote answer) '* post_act))

                         ;if a POP arc return appropriate answer
                         (cond ((equal next_node nil) (return answer))

                         ;else move to next node
                         (t (setq node (get_node next_node))
                              (setq arcs (cdr node)))))
                    ;if arc is not successful return to last decision node
                    (t (pop_stack))))

               ;if test is nil return to last decision node
               (t (pop_stack)))))

     ;pop previous position in last network and return answer
     (pop_g_stack)
     (setq return_answer return_answer))
```

Get node:

Function: To retrieve a desired node and associated arcs from the grammar network.

Input: Current node in network.

Output: The desired node and associated arcs.

```
(defun get_node (node)
     (setq answer
          (dolist (g grammar)
               (cond ((equal (car g) node) (return g)))))
     (setq answer answer))
```

(vii) POP

Function: Returns to network that called current network.

Input: A list of registers to be returned.

Output: The values of the registers given.

```
(defun POP (form)
     (setq retans nil)
     (dolist (f form)
          (setq ans (getr_register f))
          (cond ((neq ans nil) (setq retans (append retans (list ans)))))))
```

```
(cond ((equal (length retans) 1) (setq retans (car retans))))
(setq retans retans))
```

Dictionary Definition

The dictionary definition consists of a list of categories and associated words that the ATN will recognize. Formatted as

(category word1 word2 ... wordi).

Currently, the dictionary does not accept 'case' information about a word or dictionary predicates to test for certain word features (i.e., plural).

ATN Interpreter

Function: To run the ATN interpreter.

Input: Grammar file, dictionary file.

Output: Interpretation of sentences according to grammar.

```
(defun ATN (gram_file dict_file)
    (load 'arc_labels.lsp); File containing arc predicate definitions
    (initial gram_file dict_file)
    (setq answer (PUSH start_node))
    (cond ((neq word_index (length user_input)) (setq answer nil)))
    (setq answer answer)
```

Initial:

Function: To initialize all variables and read all input.

Input: Grammar file, dictionary file.

Output: Nil.

```
(defun initial (grammar_file dict_file)
    (setq grammar (read_grammar grammar_file)
    (read_dict dict_file)
    (setq user_input (read t))
    (setq stack nil)
    (setq global_stack nil)
    (setq word_index 0)
    (setq start_node 'S)
    (setq post_act nil)
    (setq registers nil)
    (setq next_node nil))
```

Read grammar:

Function: To load grammar from grammar file.

Input: Grammar file name.

Output: A list of the grammar network.

```
(defun read_grammar (grammar_file)
    (setq myport (infile grammar_file)
    (setq grammar nil)
    (setq inlist t)
    (do () ((equal inlist nil))
        (setq inlist (read myport))
        (cond ((neq inlist nil)
            (setq grammar (append1 grammar inlist)))))
    (close myport)
    (setq grammar grammar))
```

Read dictionary:

Function: To load in the dictionary.

Input: Dictionary file name.

Output: Nil.

Side Effect: Sets up a hash table to store words of the dictionary by category.

```
(defun read_dict (dict_file)
    (setq myport (infile dict_file))
    (setq dictionary (make-hash-table :size 20: test #'equal))
    (setq inlist t)
    (do () ((equal inlist nil))
        (setq inlist (read myport))
        (cond ((neq inlist nil)
            (setq category (car inlist)) ;read in category
            (setq word_list (cdr list))
            (setq ans (gethash category dictionary)) ;test if category exists

        (cond ((equal ans nil) ;add new category
            (addhash category dictionary word_list))

        ;append to existing category
        (t (addhash category dictionary (append
            (gethash category dictionary) word_list)))))))

(defun read_input ()
    (setq user_input (read t)))
```

Register Manipulating Functions

In addition to the arc label predicates, the ATN interpreter has built-in register manipulating functions.

set_register (register value)	sets register to value
append_register (register value)	appends value to register
get_register (register)	gets the value(s) of a register
clear_register (register)	deletes register and associated values

The 'value' parameter in the register predicates may be replaced with a '*' which is used in "post_actions" of an arc. In this case, the '*' or value parameter, is set to/replaced with the value returned when the arc is traversed.

The program code for the register manipulating function set_register, append_register, get_register, and clear_register is as follows.

(i) SET REGISTER:

```
(defun set_register (reg val)
    (cond ((neq register nil)
          (setq ans
               (dolist (r registers)
                    (cond ((equal reg (car r)) (return r)))))
          (cond ((equal ans nil)
               (setq registers (append1 registers (list reg val))))
          (t (setq registers (subst (list reg val) ans registers)))))
    (t (setq registers (append1 registers (list reg val))))))
```

(ii) APPEND REGISTER

```
(defun append_register (reg val)
    (cond ((neq registers nil)
          (setq ans
               (dolist (r registers)
                    (cond ((equal reg (car r)) (return r)))))
          (cond ((equal ans nil)
               (setq registers (append1 registers (list reg val))))
          (t (setq new (append ans (list val)))
               (setq registers (subst new ans registers)))))
    (t (setq registers (append1 registers (list reg val))))))
```

(iii) GET REGISTER

```
(defun get_register (reg)
    (cond ((not listp reg)) (setq reg (list reg))))
    (setq ret_ans nil)
    (dolist ((rr reg)
          (setq ans
               (dolist (r registers)
                    (cond ((equal rr (car r)) (return cdr r))))))
          (cond ((atom ans) (setq ret_ans (append ret_ans (list ans))))
```

```
        (t (setq ret_ans (append ret_ans ans)))))
    (cond ((atom ret_ans) (setq ret_ans ret_ans))
        ((equal (length ret_ans) 1) (setq ret_ans (car ret_ans)))
    (t (setq ret_ans ret_ans))))
```

(iv) CLEAR REGISTER

```
(defun clear_register (reg)
    (cond ((not listp reg)) (setq reg (list reg))))
    (setq ret_ans nil)
    (dolist (rr reg)
        (setq ans
            (dolist (r registers)
                (cond ((equal rr (car r)) (return r)))))
        (setq registers (delete ans registers 1))))
```

Backtracking

Two stacks are used for backtracking purposes. The 'global stack' is used to remember where the interpreter left when another network is invoked by a PUSH arc. The 'stack' is used to record information about the current state of the system whenever a node has more than one arc attached. This allows the system to return immediately to the last decision node if a 'dead-end' situation occurs along the current path.

The 'global stack' stores the information concerning the 'stack' of decision nodes (i.e., past decision points), the current preaction, and the next node to be traversed when returning from the subnetwork. This information is stored when the PUSH predicate is invoked. Information is popped off the global stack when a POP arc is incurred. The functions which maintain the global stack are 'push_g_stack' and 'pop_g_stack.'

The stack of decision nodes stores the values of all registers, the current input word being tested, and all remaining arcs of the node. This information is stored at any node which has more than one arc associated with it. Information is retrieved from this stack when a dead-end occurs in the current path. The functions which maintain the stack are 'push_stack' and 'pop_stack.'

Stack Manipulating Functions

The following functions are used for manipulating a last-in-first-out (lifo) stack for both local and global variables.

(i) PUSH STACK

```
(defun push_stack (registers word_index arcs)
    (setq stack (append1 stack (list registers word_index arcs))))
```

(ii) POP STACK

```
(defun pop_stack ()
    (cond ((equal stack nil) (setq arcs nil))
```

```
    (t (setq lifo (nth (- (length stack) 1) stack))
        (setq registers (car lifo))
        (setq word_index (nth 1 lifo))
        (setq arcs (nth 2 lifo))
        (setq stack (ldiff stack (member lifo stack))))))
```

(iii) PUSH GLOBAL STACK

```
(defun push_g_stack (stack post_act next_node)
    (setq global_stack (append1 global_stack (list stack post_act next_node))))
```

(iv) POP GLOBAL STACK

```
(defun pop_g_stack ()
    (cond ((neq global_stack nil)
        (setq lifo (nth (- (length global_stack ) 1) global_stack))
        (setq stack (car lifo))
        (setq post_act (nth 1 lifo))
        (setq next_node (nth 2 lifo))
        (setq global_stack (ldiff global_stack (member lifo global_stack))))))
```

COMMENTS

The development of the three AI knowledge representation tools discussed in this report are first attempts at designing knowledge-based AI systems for material testing and evaluation applications. This includes: expert systems for materials selection, classification, and qualification; intelligent robotic systems for materials testing; and machine learning techniques for automated knowledge acquisition and for the correlation and interpretation of test results.

Currently, work is being done to develop a production expert system for materials selection and qualification. Plans are to (i) utilize the frame-based system for storing information on materials, (ii) formulate new production rule predicates for accessing information from the frame-based system, (iii) build a grammar and dictionary for parsing user questions into appropriate queries for both the production rule interpreter and the frame-based system, and (iv) expand the ATN interpreter so that case information (e.g., word patterns) may be accepted and stored utilizing the frame-based environment.

DISTRIBUTION LIST

No. of Copies	To
1	Office of the Under Secretary of Defense for Research and Engineering, The Pentagon, Washington, DC 20301
	Commander, U.S. Army Laboratory Command, 2800 Powder Mill Road, Adelphi, MD 20783-1145
1	ATTN: AMSLC-IM-TL
1	AMSLC-CT
	Commander, Defense Technical Information Center, Cameron Station, Building 5, 5010 Duke Street, Alexandria, VA 22304-6145
2	ATTN: OTIC-FDAC
1	Metals and Ceramics Information Center, Battelle Columbus Laboratories, 505 King Avenue, Columbus, OH 43201
	Commander, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709-2211
1	ATTN: Information Processing Office
	Commander, U.S. Army Materiel Command, 5001 Eisenhower Avenue, Alexandria, VA 22333
1	ATTN: AMCLD
	Commander, U.S. Army Materiel Systems Analysis Activity, Aberdeen Proving Ground, MD 21005
1	ATTN: AMXSY-MP, H. Cohen
	Commander, U.S. Army Missile Command, Redstone Scientific Information Center, Redstone Arsenal, AL 35898-5241
1	ATTN: AMSMI-RD-CS-R/Doc
1	AMSMI-RLM
	Commander, U.S. Army Armament, Munitions and Chemical Command, Dover, NJ 07801
2	ATTN: Technical Library
1	AMDAR-LCA, Mr. Harry E. Pebly, Jr., PLASTEC, Director
	Commander, U.S. Army Natick Research, Development and Engineering Center, Natick, MA 01760
1	ATTN: Technical Library
	Commander, U.S. Army Satellite Communications Agency, Fort Monmouth, NJ 07703
1	ATTN: Technical Document Center
	Commander, U.S. Army Tank-Automotive Command, Warren, MI 48397-5000
1	ATTN: AMSTA-ZSK
2	AMSTA-TSL, Technical Library
	Commander, White Sands Missile Range, NM 88002
1	ATTN: STEWS-WS-VT
	President, Airborne, Electronics and Special Warfare Board, Fort Bragg, NC 28307
1	ATTN: Library
	Director, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD 21005
1	ATTN: SLCBR-TSB-S (STINFO)
	Commander, Dugway Proving Ground, Dugway, UT 84022
1	ATTN: Technical Library, Technical Information Division
	Commander, Harry Diamond Laboratories, 2800 Powder Mill Road, Adelphi, MD 20783
1	ATTN: Technical Information Office
	Director, Benet Weapons Laboratory, LCWSL, USA AMCCOM, Watervliet, NY 12189
1	ATTN: AMSMC-LCB-TL
1	AMSMC-LCB-R
1	AMSMC-LCB-RM
1	AMSMC-LCB-RP
	Commander, U.S. Army Foreign Science and Technology Center, 220 7th Street, N.E., Charlottesville, VA 22901
1	ATTN: Military Tech

No. of Copies	To
	Commander, U.S. Army Aeromedical Research Unit, P.O. Box 577, Fort Rucker, AL 36360
1	ATTN: Technical Library
	Commander, U.S. Army Aviation Systems Command, Aviation Research and Technology Activity, Aviation Applied Technology Directorate, Fort Eustis, VA 23604-5577
1	ATTN: SAVDL-E-MOS
	U.S. Army Aviation Training Library, Fort Rucker, AL 36360
1	ATTN: Building 5906-5907
	Commander, U.S. Army Agency for Aviation Safety, Fort Rucker, AL 36362
1	ATTN: Technical Library
	Commander, USACDC Air Defense Agency, Fort Bliss, TX 79916
1	ATTN: Technical Library
	Commander, U.S. Army Engineer School, Fort Belvoir, VA 22060
1	ATTN: Library
	Commander, U.S. Army Engineer Waterways Experiment Station, P. O. Box 631, Vicksburg, MS 39180
1	ATTN: Research Center Library
	Commandant, U.S. Army Quartermaster School, Fort Lee, VA 23801
1	ATTN: Quartermaster School Library
	Naval Research Laboratory, Washington, DC 20375
1	ATTN: Code 5830
2	Dr. G. R. Yoder - Code 6384
	Chief of Naval Research, Arlington, VA 22217
1	ATTN: Code 471
1	Edward J. Morrissey, AFWAL/MLTE, Wright-Patterson Air Force, Base, OH 45433
	Commander, U.S. Air Force Wright Aeronautical Laboratories, Wright-Patterson Air Force Base, OH 45433
1	ATTN: AFWAL/MLC
1	AFWAL/MLLP, M. Forney, Jr.
1	AFWAL/MLBC, Mr. Stanley Schulman
	National Aeronautics and Space Administration, Marshall Space Flight Center, Huntsville, AL 35812
1	ATTN: R. J. Schwinghammer, EH01, Dir, M&P Lab
1	Mr. W. A. Wilson, EH41, Bldg. 4612
	U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD 20899
1	ATTN: Stephen M. Hsu, Chief, Ceramics Division, Institute for Materials Science and Engineering
1	Committee on Marine Structures, Marine Board, National Research Council, 2101 Constitution Ave., N.W., Washington, DC 20418
1	Librarian, Materials Sciences Corporation, Guynedd Plaza II, Bethlehem Pike, Spring House, PA 19477
1	The Charles Stark Draper Laboratory, 68 Albany Street, Cambridge, MA 02139
	Wyman-Gordon Company, Worcester, MA 01601
1	ATTN: Technical Library
	Lockheed-Georgia Company, 86 South Cobb Drive, Marietta, GA 30063
1	ATTN: Materials and Processes Engineering Dept. 71-11, Zone 54
	General Dynamics, Convair Aerospace Division, P.O. Box 748, Fort Worth, TX 76101
1	ATTN: Mfg. Engineering Technical Library
1	Mechanical Properties Data Center, Belfour Stulen Inc., 13917 W. Bay Shore Drive, Traverse City, MI 49684
1	Mr. R. J. Zentner, EAI Corporation, 626 Towne Center Drive, Suite 205, Joppatowne, MD 21085-4440
	Director, U.S. Army Materials Technology Laboratory, Watertown, MA 02172-0001
2	ATTN: SLCMT-TML
1	Author

U.S. Army Materials Technology Laboratory
Watertown, Massachusetts 02172-0001
REPRESENTING KNOWLEDGE INTELLIGENTLY: PRODUCTION RULES, FRAMES, AND TRANSITIONAL NETWORKS - Suzanne G. W. Dunn

Technical Report MTL TR 89-94, October 1989, 42 pp-illus, D/A Project: 1L162105.AH84

AD__________________
UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

Artificial intelligence
Natural language
Frame-based knowledge

Designs for a production rule interpreter, a frame-based knowledge system, and an augmented transitional network are described. Knowledge represented in the form of production rules (consisting of domain facts and heuristics) is a good way to model the strong data-driven nature of intelligent action. Using this knowledge, production systems make inferences on the system's current understanding of the "state of the world." The production rule interpreter described uses a bottom-up approach employing a forward-chaining control strategy. Frames are complex data structures for representing stereotyped objects, events, or situations. For intelligent computer programs requiring this type of information, frame-based knowledge systems minimize redundant information and may be utilized for acquiring new information which is interpreted in terms of concepts acquired through previous experience. The frame-based system described utilizes procedural as well as declarative information. An augmented transitional network is used for natural language understanding. The transitional network described here uses phrase-structured grammars to syntactically decompose and analyze input sentences.

U.S. Army Materials Technology Laboratory
Watertown, Massachusetts 02172-0001
REPRESENTING KNOWLEDGE INTELLIGENTLY: PRODUCTION RULES, FRAMES, AND TRANSITIONAL NETWORKS - Suzanne G. W. Dunn

Technical Report MTL TR 89-94, October 1989, 42 pp-illus, D/A Project: 1L162105.AH84

AD__________________
UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

Artificial intelligence
Natural language
Frame-based knowledge

Designs for a production rule interpreter, a frame-based knowledge system, and an augmented transitional network are described. Knowledge represented in the form of production rules (consisting of domain facts and heuristics) is a good way to model the strong data-driven nature of intelligent action. Using this knowledge, production systems make inferences on the system's current understanding of the "state of the world." The production rule interpreter described uses a bottom-up approach employing a forward-chaining control strategy. Frames are complex data structures for representing stereotyped objects, events, or situations. For intelligent computer programs requiring this type of information, frame-based knowledge systems minimize redundant information and may be utilized for acquiring new information which is interpreted in terms of concepts acquired through previous experience. The frame-based system described utilizes procedural as well as declarative information. An augmented transitional network is used for natural language understanding. The transitional network described here uses phrase-structured grammars to syntactically decompose and analyze input sentences.

U.S. Army Materials Technology Laboratory
Watertown, Massachusetts 02172-0001
REPRESENTING KNOWLEDGE INTELLIGENTLY: PRODUCTION RULES, FRAMES, AND TRANSITIONAL NETWORKS - Suzanne G. W. Dunn

Technical Report MTL TR 89-94, October 1989, 42 pp-illus, D/A Project: 1L162105.AH84

AD__________________
UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

Artificial intelligence
Natural language
Frame-based knowledge

Designs for a production rule interpreter, a frame-based knowledge system, and an augmented transitional network are described. Knowledge represented in the form of production rules (consisting of domain facts and heuristics) is a good way to model the strong data-driven nature of intelligent action. Using this knowledge, production systems make inferences on the system's current understanding of the "state of the world." The production rule interpreter described uses a bottom-up approach employing a forward-chaining control strategy. Frames are complex data structures for representing stereotyped objects, events, or situations. For intelligent computer programs requiring this type of information, frame-based knowledge systems minimize redundant information and may be utilized for acquiring new information which is interpreted in terms of concepts acquired through previous experience. The frame-based system described utilizes procedural as well as declarative information. An augmented transitional network is used for natural language understanding. The transitional network described here uses phrase-structured grammars to syntactically decompose and analyze input sentences.

U.S. Army Materials Technology Laboratory
Watertown, Massachusetts 02172-0001
REPRESENTING KNOWLEDGE INTELLIGENTLY: PRODUCTION RULES, FRAMES, AND TRANSITIONAL NETWORKS - Suzanne G. W. Dunn

Technical Report MTL TR 89-94, October 1989, 42 pp-illus, D/A Project: 1L162105.AH84

AD__________________
UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

Artificial intelligence
Natural language
Frame-based knowledge

Designs for a production rule interpreter, a frame-based knowledge system, and an augmented transitional network are described. Knowledge represented in the form of production rules (consisting of domain facts and heuristics) is a good way to model the strong data-driven nature of intelligent action. Using this knowledge, production systems make inferences on the system's current understanding of the "state of the world." The production rule interpreter described uses a bottom-up approach employing a forward-chaining control strategy. Frames are complex data structures for representing stereotyped objects, events, or situations. For intelligent computer programs requiring this type of information, frame-based knowledge systems minimize redundant information and may be utilized for acquiring new information which is interpreted in terms of concepts acquired through previous experience. The frame-based system described utilizes procedural as well as declarative information. An augmented transitional network is used for natural language understanding. The transitional network described here uses phrase-structured grammars to syntactically decompose and analyze input sentences.